

PATENT
81940.0060
Express Mail Label No. EV 324 110 834 US

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:

Hiroyasu NISHIYAMA

Serial No: Not assigned

Filed: October 29, 2003

For: Interpreter And Native Code Execution Method

Art Unit: Not assigned

Examiner: Not assigned

TRANSMITTAL OF PRIORITY DOCUMENT

Mail Stop PATENT APPLICATION

Commissioner for Patents

P.O. Box 1450

Alexandria, VA 22313-1450

Dear Sir:

Enclosed herewith is a certified copy of Japanese patent application No. 2003-097574 which was filed April 1, 2003, from which priority is claimed under 35 U.S.C. § 119 and Rule 55.

Acknowledgment of the priority document(s) is respectfully requested to ensure that the subject information appears on the printed patent.

Respectfully submitted,

HOGAN & HARTSON L.L.P.

Date: October 29, 2003

By: 

Anthony J. Orler
Registration No. 41,232
Attorney for Applicant(s)

500 South Grand Avenue, Suite 1900
Los Angeles, California 90071
Telephone: 213-337-6700
Facsimile: 213-337-6701



日 本 国 特 許 庁
JAPAN PATENT OFFICE

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office.

出 願 年 月 日 2 0 0 3 年 4 月 1 日
Date of Application:

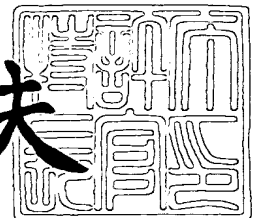
出 願 番 号 特 願 2 0 0 3 - 0 9 7 5 7 4
Application Number:
[ST. 10/C] : [J P 2 0 0 3 - 0 9 7 5 7 4]

出 願 人 株式会社日立製作所
Applicant(s):

2 0 0 3 年 9 月 3 0 日

特許庁長官
Commissioner,
Japan Patent Office

今 井 康 夫



【書類名】 特許願

【整理番号】 K03001171A

【あて先】 特許庁長官殿

【国際特許分類】 G06F 9/45

【発明者】

【住所又は居所】 神奈川県川崎市麻生区王禅寺 1 0 9 9 番地 株式会社日立製作所 システム開発研究所内

【氏名】 西山 博泰

【特許出願人】

【識別番号】 000005108

【氏名又は名称】 株式会社日立製作所

【代理人】

【識別番号】 100075096

【弁理士】

【氏名又は名称】 作田 康夫

【手数料の表示】

【予納台帳番号】 013088

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【プルーフの要否】 要



【書類名】 明細書

【発明の名称】 インタープリタおよびネイティブコード実行方法

【特許請求の範囲】

【請求項 1】

ネイティブコードの呼び出し機能を備え、処理装置と協働してプログラミング言語の実行を行うインタープリタであって、ネイティブコードをネイティブコードエミュレータによるハードウェアエミュレーションによって実行することを特徴とするインタープリタ。

【請求項 2】

請求項 1 において、前記ネイティブコードによるメモリ参照命令の監視を行うことを特徴とするインタープリタ。

【請求項 3】

請求項 2 において、前記メモリ参照命令の監視は、当該インタープリタの管理するメモリ領域に関して、ネイティブコードからの読み出し/書き込み/実行の可/不可の表を記録しておき、前記ネイティブコードエミュレータによるネイティブコードの実行時に前記表を参照して命令実行における不正参照検出を実施することを特徴とするインタープリタ。

【請求項 4】

請求項 1 において、プログラムの実行対象コードが、インタープリタコードとネイティブコードのいずれに属するかを判定し、前記実行対象コードがネイティブコードであると判定された場合にエミュレータにて処理を行うことを特徴とするインタープリタ。

【請求項 5】

請求項 4 において、ネイティブメソッド呼び出しにてインタープリタコードの実行とネイティブコードの実行の遷移が行われる場合に、ネイティブメソッド呼び出しが発生するまでは前記判定処理を実行しないことを特徴とするインタープリタ。

【請求項 6】

ネイティブコードの呼び出し機能を備え、処理装置と協働してプログラミング

言語の実行を行うインタプリタであって、ネイティブコードによるメモリ参照命令の監視処理を行うことを特徴とするインタプリタ。

【請求項 7】

請求項 6 において、前記メモリ参照命令の監視は、当該インタプリタの管理するメモリ領域に関して、ネイティブコードからの読み出し/書き込み/実行の可/不可の表を記録しておき、前記ネイティブコードエミュレータによるネイティブコードの実行時に前記表を参照して命令実行における不正参照検出を実施することを特徴とするインタプリタ。

【請求項 8】

ネイティブコードの呼び出し機能を備え、処理装置と協働してプログラミング言語の実行を行うインタプリタにおいて、ネイティブコード部分をハードウェアで直接実行するのではなく、ネイティブコードエミュレータによるハードウェアエミュレーションによって実行することを特徴とするネイティブコード実行方法。

【請求項 9】

請求項 8 において、インタプリタからネイティブコードの呼出を実施する際に、インタプリタの管理する各々のメモリ領域に関して、ネイティブコードからの読み出し/書き込み/実行の可/不可の表を記録しておき、前記ネイティブコードエミュレータによってネイティブコードの実行を行う際に、前記表を参照して、命令実行における不正参照検出を実施するネイティブコード実行方法。

【請求項 10】

請求項 1 のインタプリタにおいて、前記ネイティブコード部分の実行状態をネイティブコードエミュレータ内部に記憶しておき、プログラムの実行状態の退避を行う際に、インタプリタの内部状態と共にネイティブコード部分の実行状態を退避することを特徴とするインタプリタ。

【請求項 11】

請求項 10 のインタプリタにおいて、退避したプログラムの実行状態を読み出し、プログラムの停止点からプログラム実行を再開することを特徴とするインタプリタ。

【発明の詳細な説明】**【0001】****【発明の属する技術分野】**

本発明はネイティブコードの呼び出し機能を持つインタプリタ型の実行を行うプログラミング言語におけるプログラムの実行方法に関する。

【0002】**【従来の技術】**

プログラミング言語の実行方法として、コンパイラと呼ばれるプログラムによってソースプログラムを実行対象計算機の機械語コードに変換する方法と、ソースプログラム、あるいは、それを中間表現に変換したプログラムを、インタプリタと呼ばれるプログラムによって、解釈実行する方式がよく知られている。

【0003】

インタプリタは、別のプログラムを解釈実行するプログラムの総称であり、プログラムの可搬性を高めるためにインタプリタによる実行方式が採用されることが多い。インタプリタ型の言語では、プログラムの実行がインタプリタと呼ばれるプログラムによって行われるため、インタプリタ実行されるプログラムで発生した不正メモリ参照などの問題を検出することが容易である。このようなプログラミング言語の代表として、Java (R) (非特許文献1: J. Gosling 他, Java Language Specification, Sun Microsystems, 2000.) を挙げることができる。Javaで記述されたアプリケーションは、一旦バイトコードと呼ばれる中間表現に変換されJava仮想マシンと呼ばれるソフトウェアによってバイトコードが解釈実行される。

【0004】

Javaでは、配列外メモリ参照、nullポインタ参照などを実行時に検出する機能を持ち、不正なメモリ破壊が生じ得ないことを特徴としている。一方、C言語やC++などネイティブコード実行を行うプログラムでは、言語の機能としてメモリを保護する機能を備えていないため、OSが保護を行っている領域外のメモリを不正に参照することができる。

【0005】

こういった、Javaなどのインタープリタ型言語では、一般にI/Oなど低レベルのライブラリ機能をそれ自身で記述することが難しい。そこで、こういった低レベル機能に関しては、C言語やC++などによって記述されたネイティブコードを利用して実装する方式が広く用いられている。例えば、Javaではユーザがプログラム中から呼び出したいメソッドをネイティブコードによって記述するためのJNI(非特許文献2:S. Liang, Java Native Interface: Programmer's Guide and Specification, Sun Microsystems, 1999.)と呼ばれる仕様を定めている。JNIでは、Javaからネイティブコードの呼び出しだけでなく、ネイティブコードからJavaプログラムを呼び出すための仕様も定めている。

【0 0 0 6】

一方、このようなプログラムの実行において、プログラムそのものやハードウェアの保守・管理などの目的で、実行中のアプリケーションプログラムを一時的に中断してファイル上に退避しておき後ほど実行したい場合がある(checkpoint/restart)。また、同様の目的で、あるハードウェア上で実行しているアプリケーションを別のハードウェアに移動して実行継続したいというケース(migration)がある。このようなケースでは、プログラムの実行中の状態を取り出して、退避・回復できる必要がある。

【0 0 0 7】

Javaにおいてこのようなプログラムの実行状態の退避・回復を実現するための方式として、バイトコードを解析し、退避を行いたいプログラム点に関して、スタック上の値などの情報を取得するためのコードをバイトコード列に挿入し、プログラム実行状態の回復を行う場合は、退避したプログラム状態を再構成するようにバイトコードを変換する方式が知られている(非特許文献3:E. Truyen他、Portable Support for Transparent Thread Migration in Java, In Proceedings of International Symposium on Agent Systems and Applications/Mobile Agents, 2000)。

【0 0 0 8】

【非特許文献1】 J. Gosling他, Java Language Specification, Sun Microsystems, 2000

【非特許文献 2】 S. Liang, Java Native Interface: Programmer's Guide and Specification, Sun Microsystems, 1999

【非特許文献 3】 E. Truyen他、Portable Support for Transparent Thread Migration in Java, In Proceedings of International Symposium on Agent Systems and Applications/Mobile Agents, 2000

【 0 0 0 9 】

【発明が解決しようとする課題】

ネイティブコードによってプログラムの一部を実装する機能は、インタプリタのみで実装できない機能を実現するために必須であるが、次のような問題がある。

(1) 安全性の保障が不十分

ネイティブコード部分のプログラムからは、インタプリタ内のメモリを自由に参照可能である。よって、インタプリタ部分で不正メモリ参照の検査を行っているとしても、ネイティブコード部分のプログラムに誤りがあった場合、プログラムの安定性/安全性が保証できなくなる。

(2) 状態の退避・回復が困難

インタプリタ実行を行っている部分に関しては、インタプリタがプログラムの実行途中の状態(メモリ上のデータ値、実行中の命令アドレスなど)を管理しているため、実行状態を退避・回復することは容易である。一方、インタプリタから呼び出されたネイティブコード実行部分に関しては、インタプリタの管理外で実行が行われているため、実行状態を退避・回復することが困難である。

【 0 0 1 0 】

例えば、上記の従来技術では、プログラムの実行状態の退避・回復は純粋なJava実行を行っているアプリケーションのみに限られている。一般には、Javaのライブラリ中にはネイティブコードとして実現されている部分が多数存在するため、プログラム実行状態の退避・回復を行う上での大きな制約となる。

【 0 0 1 1 】

【課題を解決するための手段】

上記の問題は、ネイティブコード部分の実行がインタプリタの制御とは無関

係に行なわれることによる。そこで、本発明ではインタプリタから呼び出されるネイティブコード部分に関しては、ハードウェアで直接実行を行うのではなく、ハードウェアの機能のエミュレーション実行を行うエミュレータにより実行するようにした。こうすることでネイティブコードの処理をインタプリタによりコントロールすることが可能になる。

【0012】

このエミュレータにより、ネイティブコード部分でのメモリ参照のチェックを行うことにより、ネイティブコード実行における不正メモリ参照の発生を検出することが可能となる。

【0013】

同様に、ネイティブコード部分のプログラム実行状態に関し、エミュレータによって状態変化を記録しておくことで、実行状態の退避・回復を行うことが可能となる。

【0014】

【発明の実施の形態】

以下、本発明の一実施例を図面を参照しながら説明する。

【0015】

図11は本実施例でインタプリタを実行するシステムのハードウェア構成例を示す。改良されたインタプリタは、ディスク装置1103から主記憶装置1103上に読み込まれ、プロセッサ1101上で実行される。インタプリタが実行するアプリケーションに関しても同様にディスク装置1103に格納され、主記憶1103に読み込まれ、プロセッサ 1101上で動作するインタプリタにより実行が行なわれる。インタプリタはFDやCD-R等の記憶媒体に格納されて取引され、実行前に図示していない媒体読取装置から読み込まれてディスク装置1103に格納されているものとする。

【0016】

図2は従来のインタプリタシステムの例である。アプリケーションプログラム101は、インタプリタコード102と、ネイティブコード103から構成される。インタプリタコード102はプロセッサ106上でインタプリタ104により解釈実

行される。

【 0 0 1 7 】

図1は本発明を適用したインタプリタシステムを模式的に示した図である。アプリケーションプログラム101は、インタプリタコード102と、ネイティブコード103から構成される。インタプリタコード102はプロセッサ106上でインタプリタ104により従来から行われている方法で解釈実行される。一方、ネイティブコード103は、ネイティブコードエミュレータ105を介して実行される。ネイティブコードをエミュレータにより実行する場合は、プロセッサ上で直接実行する場合と比較して、実行速度が低下する可能性があるが、エミュレーションを高速に行うための技術として、実行時に動的に機械語コードの生成を行うバイナリトランスレーションなどの技術を利用することにより、性能の低下を低く押えることも可能である。以下の実施例ではインタプリタがネイティブコードエミュレータを備えた例で説明するが、インタプリタにはネイティブコードエミュレータの呼び出し機能だけを備えさせて、インタプリタとネイティブコードエミュレータがそれぞれ独立したプログラムとしてもよい。

【 0 0 1 8 】

図3に本実施例のインタプリタによるプログラム実行処理のフローを示す。プログラム実行が処理301で開始されると、処理302で実行対象のコードがインタプリタコードであるか、ネイティブコードであるかの判定を行う。インタプリタコードである場合は、処理303に制御を移し、処理対象のコードをメモリから読み込む。次に、処理304において読み込んだインタプリタコードを実行し、処理305でプログラムの実行が完了したかどうかを判定する。プログラムの実行が完了であれば、処理306に制御を移し、プログラムの実行を終了する。完了で無い場合は、処理302に戻り次のコードの実行を行う。

【 0 0 1 9 】

処理302における判定において、当該コードがネイティブコードである場合は、処理307に制御を移し、ネイティブコードでのメモリ領域参照の可/不可の情報を表す領域表を作成する。次に、処理308に制御を移す。処理308では、ネイティブコードをメモリから読み出し、上記307で作成した領域表を参照してメモリ参

照の可/不可などの検査を付加的に行う。検査の結果参照エラーがなければ処理309において、処理308で読み出したネイティブコードの実行を行う。続いて、処理310においてプログラムの実行が完了したか否かを判定し、処理が完了していれば処理306に制御を移してプログラムの実行を終了する。処理が完了していなければ、処理302に戻って次のコードを実行する。

【 0 0 2 0 】

なお、図3のフローでは、領域表の生成はネイティブコードの実行前に行うこととしたが、プログラムの起動時に初期設定しておき、プログラムの実行に伴ってネイティブコードから参照可能な領域が変化する毎に更新する方式にしてもよい。

【 0 0 2 1 】

ここで、領域表の生成をネイティブコードの実行前に行う方式は、領域表に利用するメモリをネイティブコードの呼び出しを行わない場合は必要としないという利点があるが、一回あたりのネイティブコード呼び出しのオーバーヘッドが大きくなる。逆に、プログラムの実行に伴って領域表を構築する後者の方式では、一回あたりのネイティブコード呼び出しオーバーヘッドは低くなるが、領域表のデータをプログラム実行中に保持しておかなくてはならない。

【 0 0 2 2 】

また、図3のフローでは、各インタープリタコードおよびネイティブコードに関して、次の実行対象コードがいずれに属するかを各命令の実行毎に確認している例を示したが、インタープリタ実行とネイティブコード実行の間の遷移が特定の命令で実施される場合など、次の対象命令の種別が特定可能な場合には、これを省略することが可能である。例えば、Javaではインタープリタからネイティブコードへの遷移はネイティブメソッド呼び出しに限定されるので、ネイティブメソッド呼び出しが発生するまでは、インタープリタ実行であることを仮定し、処理302の検査を省略できる。

【 0 0 2 3 】

本実施例の特徴的な機能である、ネイティブコード実行部分におけるネイティブプログラム実行時の安全性の保障、プログラム実行状態の退避・回復のための

状態の記録は主に処理308において実施される。

【 0 0 2 4 】

図4は図3の処理308のネイティブコード命令読み出し処理で不正なメモリ参照や破壊の検出処理を実施する例の詳細を示す。まず、処理401で処理を開始し、処理402において変数Iに実行対象の命令を求める。次に、実行対象の命令Iがメモリ参照命令であるか否かを確認する。メモリ参照命令でない場合は、処理410に制御を移して処理を完了する(即ち、ステップ308の処理を終えステップ309に移ってネイティブコードの実行を行う)。メモリ参照命令の場合は、処理404に制御を移し、変数Aに命令Iが参照するアドレスを、変数Tに参照可能なメモリ領域の情報を表す領域表を求める。次に、処理405で領域表が空か否かを確認する。空の場合は、対応するアドレスが登録されていないので、処理408に制御を移して参照エラーを報告する。空でない場合は、処理406に制御を移し、領域表Tからエントリを1つ取り出し、変数Rに格納する。次に、領域情報Rに関して、アドレスAがRに含まれるか否かを確認する。参照しなければ処理405に制御を移し、次の領域の検査を継続する。参照を行う場合は、処理408に制御を移し、アドレスAの参照が不正か否かを確認する。参照が不正であれば、処理408に制御を移し処理を終了する。不正でなければ処理410に制御を移して処理を完了する。

【 0 0 2 5 】

上記処理で利用するメモリ領域に関する情報を定義する領域表は、インタプリタの実行状態から定義される。Javaインタプリタの場合であれば、ネイティブコード実行部分においては、インタプリタ管理下にあるメモリ領域に対する読み書きを行う事は基本的にできない。ただし、JNI関数を介した場合には、読み書き可能である。

【 0 0 2 6 】

図5に領域表の例を示す。表の各エントリは、開始アドレス501、終了アドレス502、参照可能モード503の3つのエントリからなる。なお、図5の参照モードにおいて、「r」は読み出し可、「w」は書き込み可を表すものとする。従って「--」は読み出しも書き込みも不可を表す。例えば、ストア命令がアドレス00310000に対して書き込みを行う場合を考える。この場合、図4の処理405～407によって、

図5の領域表のエントリ504～506が順に検査され、エントリ506でアドレス00310000の所属する領域が書き込み不可なので、処理408でエラーが報告される。また、次に、ロード命令がアドレス00220000から読出しを行う場合を考える。先の例と同様に、図4の処理405～407によって、図5の領域表のエントリ504～506が順に検査され、領域が読み出し可能であるので処理409で正常に処理が終了する。

【 0 0 2 7 】

この検査は、すべてのメモリ参照に関して行う必要は無く、不正参照でないことが自明であるケースあるいは冗長な検査を除くことも可能である。

【 0 0 2 8 】

このような不正参照を行うJavaプログラムの例を図6に示す。図6(a)のJavaプログラムでは、ネイティブメソッドfooにオブジェクトのリファレンス(アドレス)を受け渡している。図6(b)のnativeメソッドでは、受け渡されたオブジェクトのアドレスをint型のポインタにキャストし、(1)においてオブジェクトへ不正なオブジェクトへの書き込みを行っている。書き込み対象のオブジェクトはインタープリタの管理下にあるため、上記のようにネイティブコード部分からは参照できない。よって、ネイティブコード実行部分においてエラー検出が行なわれる。

【 0 0 2 9 】

次に、プログラム実行状態の退避・回復の実施例について説明する。

【 0 0 3 0 】

図7は状態退避の処理を示している。状態退避では、処理701で処理を開始する。次に、処理702でインタープリタの実行状態の退避を行ない、処理703でネイティブ実行部分の実行状態を退避し、処理704で退避処理を完了する。図8は退避した状態の回復処理を表している。回復処理も退避処理と同様の手順であり、まず処理801で処理を開始、次に処理802でインタープリタの実行状態の退避を行ない、続いて処理803でネイティブ実行部分の実行状態を回復し、処理705で処理を終了する。

【 0 0 3 1 】

図12に処理703で退避するための情報を記録する命令エミュレーション処理を示す。命令エミュレーション処理は、処理1201で処理を開始し、続いて処理1202

で実行対象の命令を変数Iに求める。次に、処理1203では命令Iが更新する状態集合を変数Tに求める。一般に、命令Iが更新する状態集合はエミュレーション対象のプロセッサの命令仕様により定義される。次に、処理1204により命令Iをエミュレーション実行する。これにより、処理1203で求めた状態集合が更新される。命令の実行後、処理1205で状態集合Tが空か否かを確認、空であれば処理1206に制御を移し処理を完了する。Tが空でなければ、処理1207に制御を移し、状態集合Tから状態を1つ取り出す。続いて、処理1208で取り出した状態の更新値を状態表に記録する。これにより、ネイティブコード実行により変更される状態の最新値が状態表に記録されることになる。続いて、処理1205に制御を移し、次の状態の処理を継続する。なお、図12に示した例では、状態表への保存を1命令の実行毎に行っているが、状態表への記録は最終値のみでよいので、複数命令を一括して処理してもよい。

【 0 0 3 2 】

次に、処理703の状態の退避処理の詳細を図13に示す。図13に示す処理は、処理1301で処理を開始し、処理1302で状態表を変数Tに求める。次に、処理1303で状態表がからかい中を確認する。空であれば処理1306に制御を移して処理を完了する。空でなければ、処理1304に制御を移し、状態表Tからエントリを1つ取り出し、変数rに格納する。続いて、処理1305で状態の識別子と状態表に格納されたその最新値を退避する。次に、処理1303に制御を移して次の状態を処理する。

【 0 0 3 3 】

図8の処理803の状態の回復処理では、この逆に、退避した状態表を読み出して、状態識別子の示す状態の最新値とすればよい。

【 0 0 3 4 】

以上の処理により、ネイティブコード実行時の最新状態を状態表に記憶できるようになり、ネイティブコード実行部分に関しても状態の退避・回復が可能となる。

【 0 0 3 5 】

図9に退避/回復の対象とするJavaプログラムの例を示す。この例では、図9(a)のJavaプログラムから図9(b)のネイティブコードを呼び出している。ここで、Ja

vaコードのループの3度目、ネイティブコード部分のループの10度目の繰返しの開始時点でプログラムの実行状態を退避/回復することを考える。

【0 0 3 6】

この時退避される情報の例を図10に示す。なお、この例では簡単のため変数の値のみを示しているが、実際にはプログラムの実行アドレスなど種々の情報を付加する必要がある。ここで、従来方式では、インタプリタ部分の変数の値に関しては停止時点での値を検出することが可能であるが、ネイティブコード部分に関しては、状態が不明であった。ネイティブコード部分の実行をエミュレータによって実施することにより、図10に示すような状態値を退避することが可能となる。プログラムの状態回復を行う場合は、図10の値を適宜読み込んで状態の回復を行えば良い。

【0 0 3 7】

【発明の効果】

本発明によれば、インタプリタ実行されるプログラムから呼び出されるネイティブコードによって生じる不正メモリ参照を検出できる。またネイティブコードの呼び出しを伴うプログラムの実行状態を退避・回復することが可能となる。

【図面の簡単な説明】

【図 1】 本発明を適用したインタプリタシステムを模式的に示した図。

【図 2】 従来のインタプリタシステムを模式的に示した図。

【図 3】 本実施例のインタプリタによるプログラム実行処理のフローを示す図。

【図 4】 図3の処理308の詳細を示すフローチャート。

【図 5】 領域表の例を示す図。

【図 6】 プログラム例を示す図。

【図 7】 状態退避の処理を示すフローチャート。

【図 8】 退避した状態の回復処理を示すフローチャート。

【図 9】 退避/回復の対象とするJavaプログラムの例を示す図。

【図 1 0】 退避された情報の例を示す図。

【図 1 1】 インタプリタを実行するシステムのハードウェア構成を示す図。

【図 1 2】 命令エミュレーション時の状態退避処理を示すフローチャート。

【図 1 3】 図7の処理703の詳細を示すフローチャート。

【符号の説明】

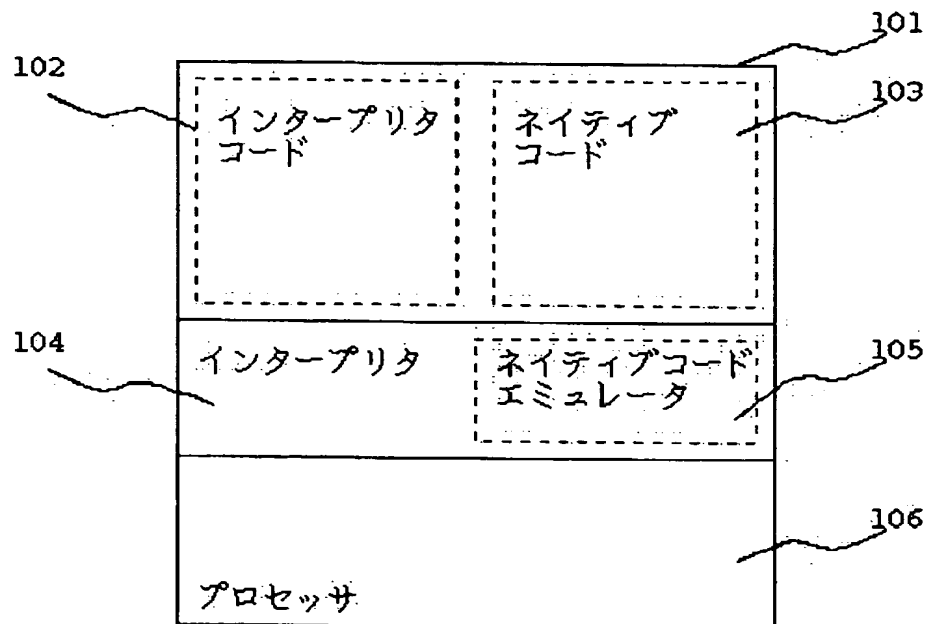
101…アプリケーションプログラム、102…インタプリタコード、103…ネイティブコード、

104…インタプリタ、106…プロセッサ。

【書類名】 図面

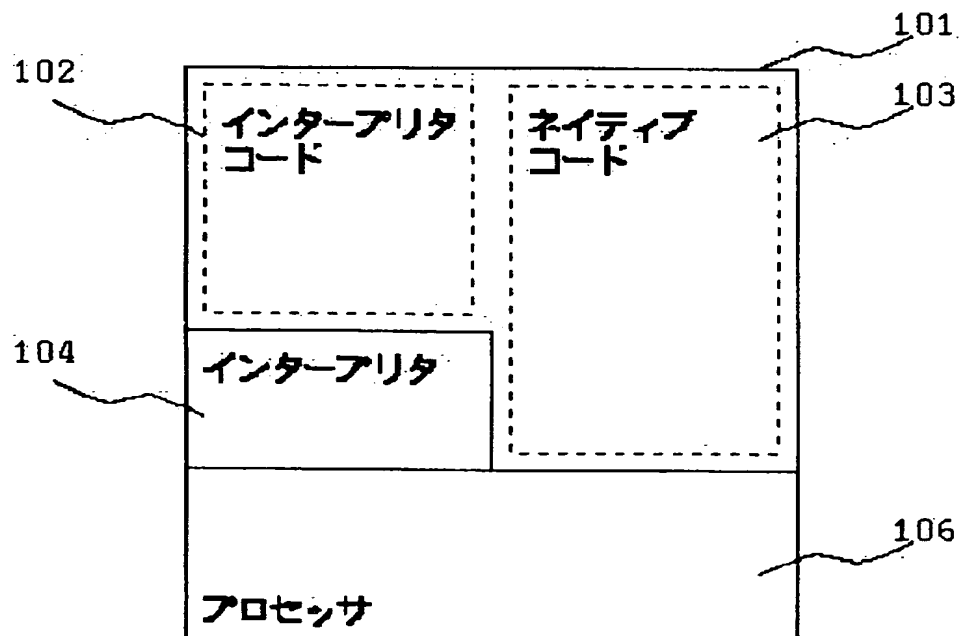
【図 1】

図 1



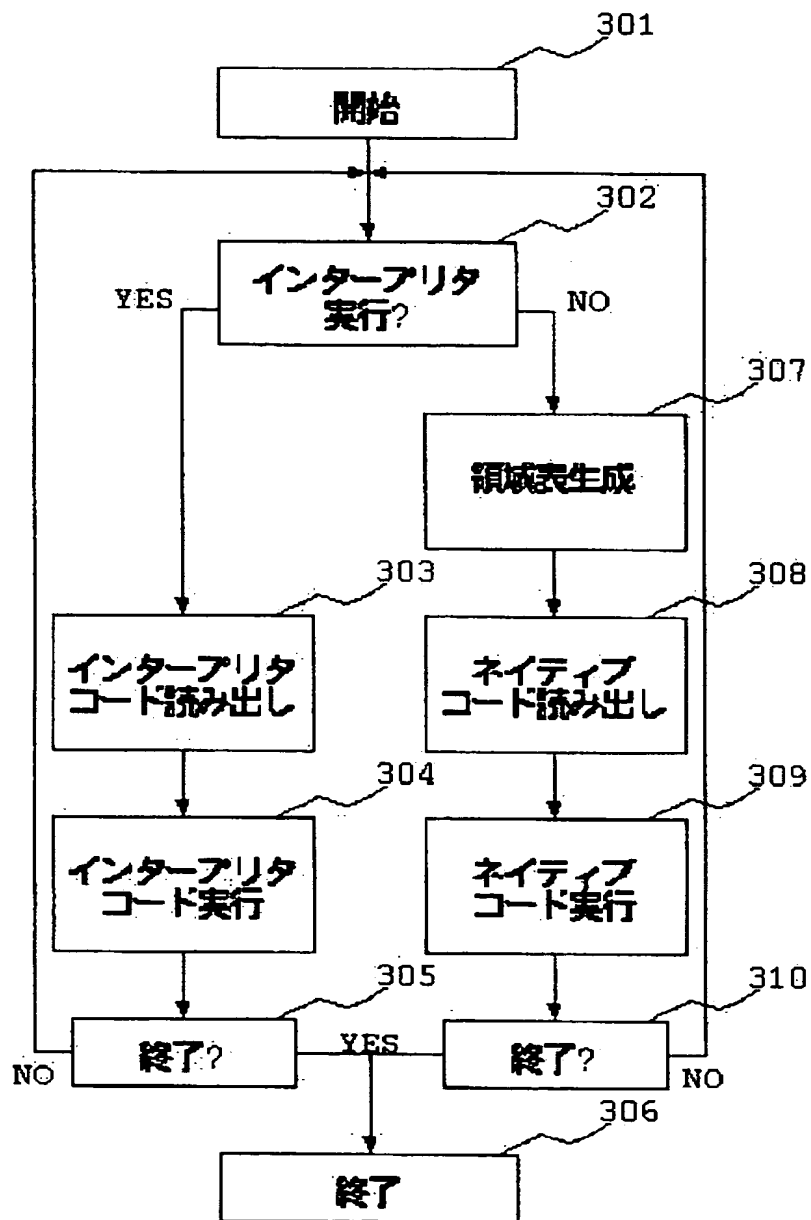
【図 2】

図 2



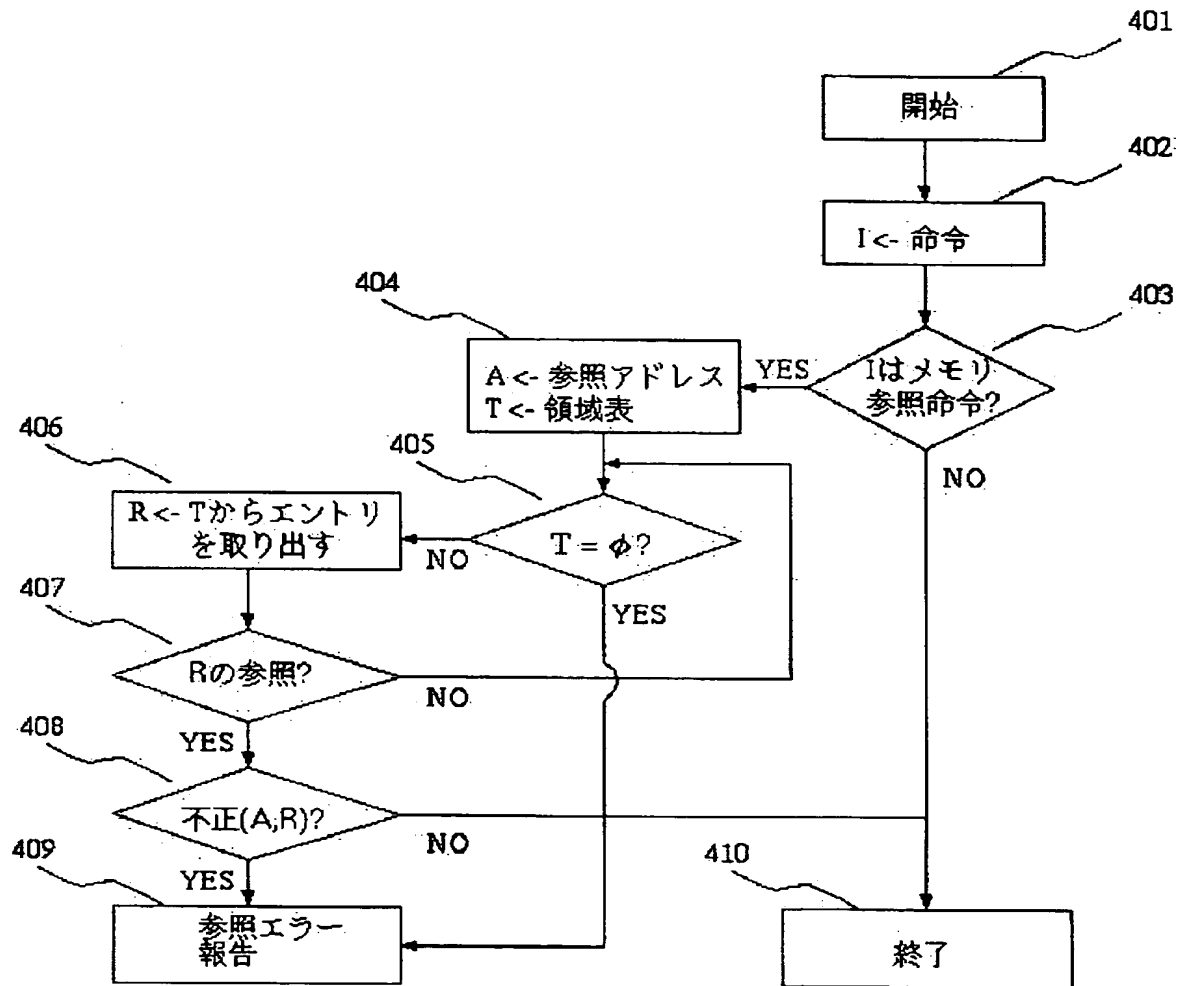
【図 3】

図 3



【図 4】

図 4



【図5】

図5

00100000	001FFFFFF	rw	501	502	503	504
00200000	002FFFFFF	r-				505
00300000	003FFFFFF	--				506

【図6】

図6

```

class C {
    static native void foo(Object obj);
    static void bar(Object obj) {
        foo(obj);
    }
}

```

(a) Javaコード

```

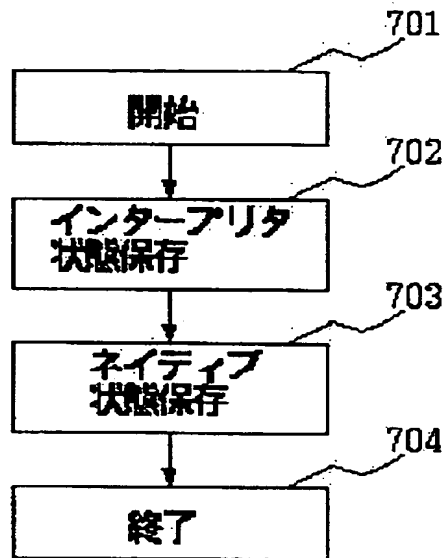
JNIEXPORT void JNICALL Java_C_foo(
    JNIEnv *env, jclass cls, jobject obj) {
    int *p = (int *)obj;
    *p = 0;                                // (1)
}

```

(b) nativeコード

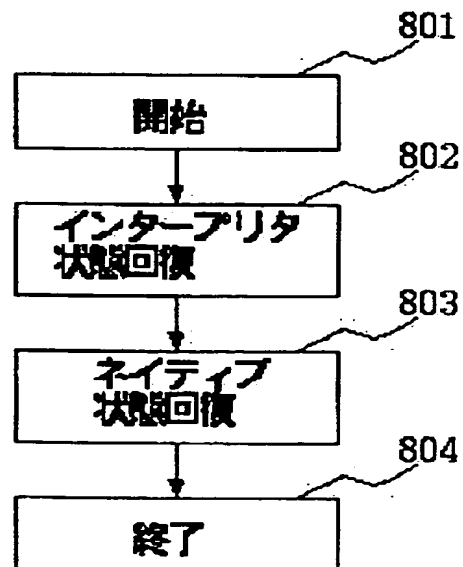
【図 7】

図 7



【図 8】

図 8



【図 9】

図 9

```
class C {  
    static native void foo();  
    static void bar() {  
        for(int j=0; j < 10; j++) {  
            foo();  
        }  
    }  
}
```

(a) Javaコード

```
JNIEXPORT jint JNICALL Java_C_foo(  
    JNIEnv *env, jclass cls) {  
    jint s = 0;  
    for(int i = 0; i < 100; i++) {  
        s = s + 1;  
    }  
    return(s);  
}
```

(b) nativeコード

【図 10】

図 10

インタープリタ

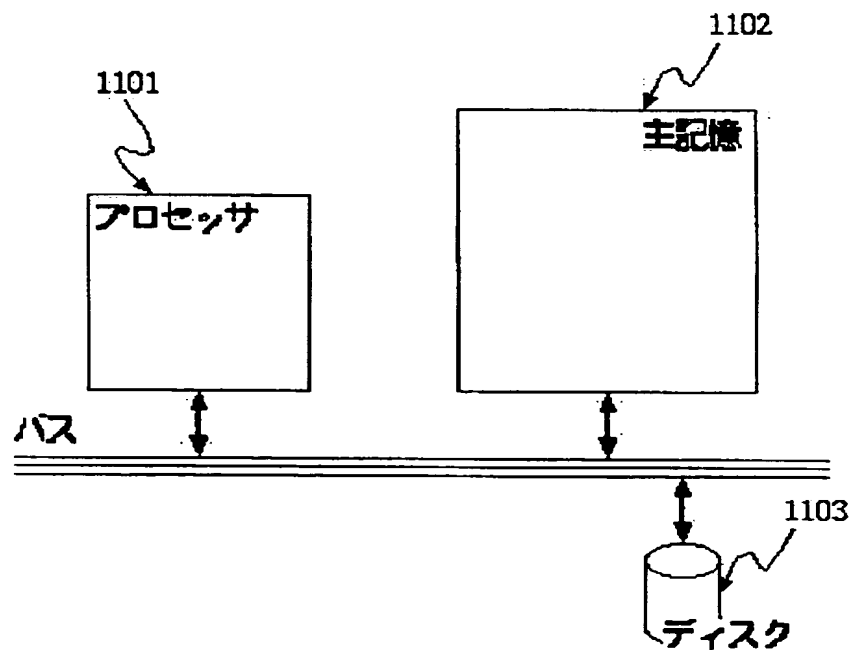
変数	値
j	3
PC	...

ネイティブコード

変数	値
i	10
s	45
PC	...

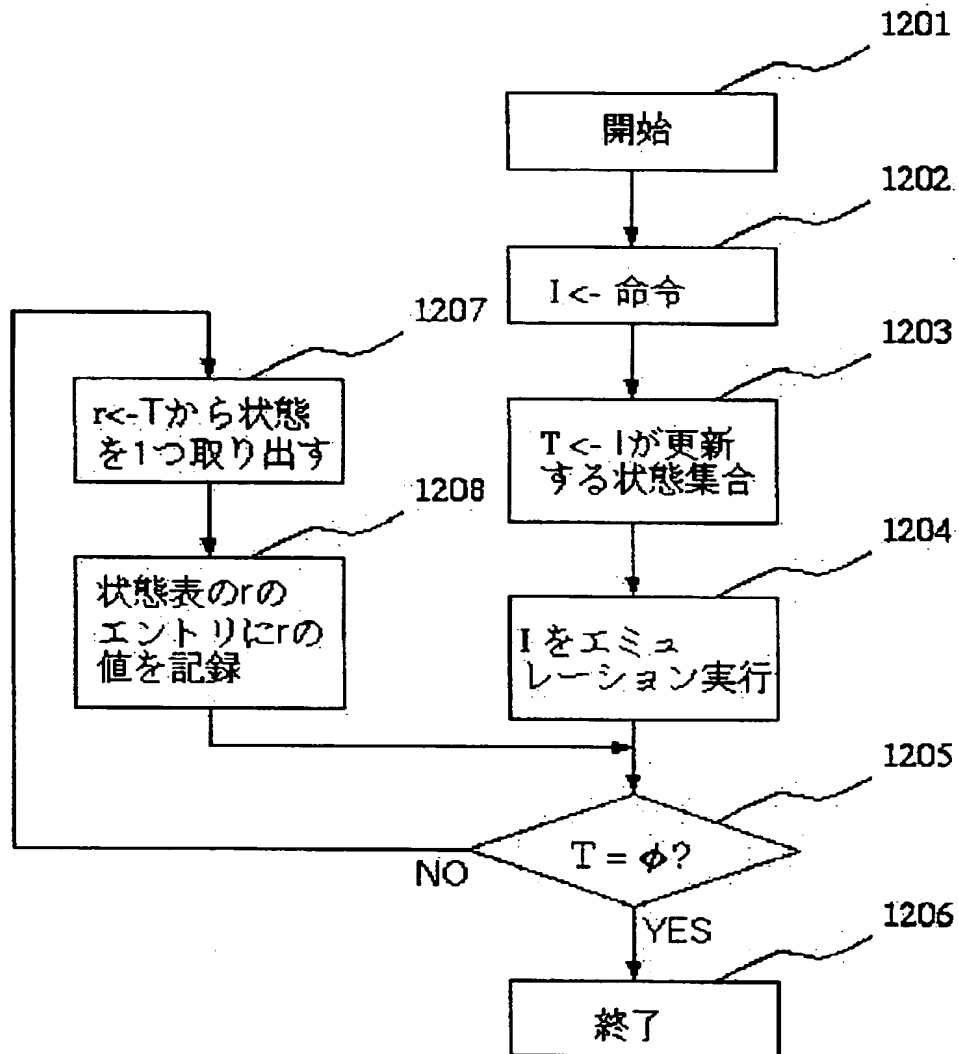
【図 11】

図 11



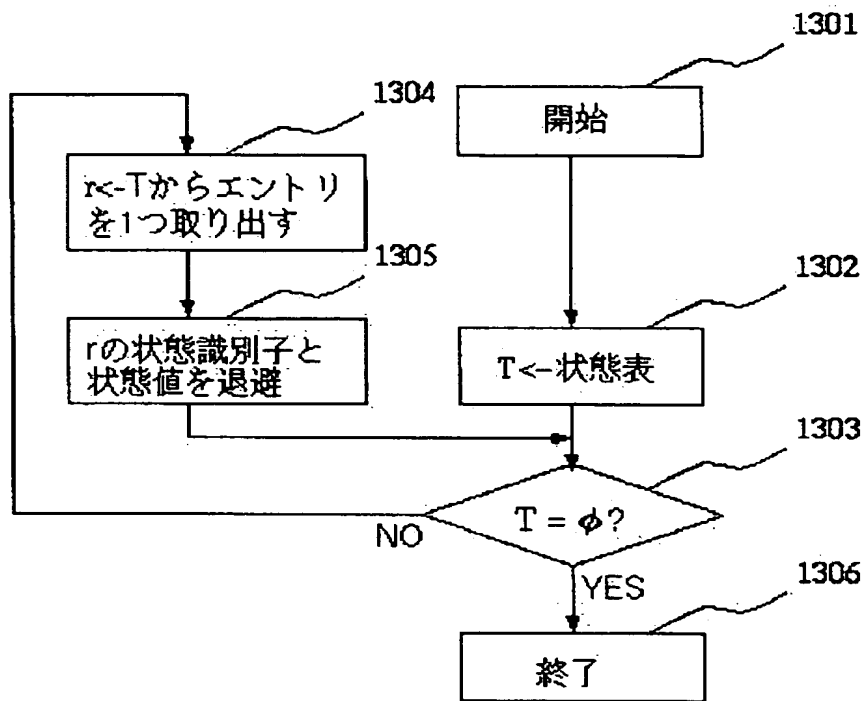
【図 12】

図 12



【図 13】

図 13



【書類名】 要約書

【要約】

【課題】

ネイティブコードの呼び出し機能を備える従来のインタプリタ型システムでは、ネイティブコードの実行に関してインタプリタの制御が及ばないため、ネイティブコード部分からの不正アクセスが可能であった。また、ネイティブコード部分の実行状態を退避・回復することも困難であった。

【解決手段】

上記課題を解決するため、インタプリタから呼び出されるネイティブコードを、実ハードウェアのエミュレーションを行うエミュレーションレイヤによって行う。エミュレーションレイヤによって不正参照のチェックや、退避・回復に必要となる実行状態の管理を行うことにより、上記課題が解決される。

【選択図】 図1

認定・付加情報

特許出願の番号	特願 2 0 0 3 - 0 9 7 5 7 4
受付番号	5 0 3 0 0 5 3 9 0 2 5
書類名	特許願
担当官	第七担当上席 0 0 9 6
作成日	平成 1 5 年 4 月 2 日

< 認定情報・付加情報 >

【提出日】	平成15年 4月 1日
-------	-------------

次頁無

特願 2 0 0 3 - 0 9 7 5 7 4

出 願 人 履 歴 情 報

識別番号

[0 0 0 0 0 5 1 0 8]

1 . 変更年月日

1 9 9 0 年 8 月 3 1 日

[変更理由]

新規登録

住 所

東京都千代田区神田駿河台 4 丁目 6 番地

氏 名

株式会社日立製作所